198:431 Software Engineering Project Information

August 26, 2010

1 Introduction

The central concerns of Software Engineering in this course are building: LARGE programs, requiring teams to build, CORRECT programs (ones that run properly *and*, especially, meet the needs of the users), EASILY MODIFIED programs. Much of this can only be learned by experience, and we think that it is better for you to learn from mistakes in a course than on the job. This handout describes a project that you will undertake with other students in the course. These notes are heavily based on ideas presented by Dr.Elaine Kant at Carnegie-Mellon University (See ACM Software Engineering Notes, Vol.6, No.4, August 1981.) One of the key benefits of this documents is that it has sprinkled throughout comments from former students who had taken Professor Kant's course.

1.1 Prerequisites

The course will be time consuming, so you must have satisfied the prerequisites listed below.

You must have taken an introductory programming course and a course in the fundamental structures of computer science (e.g., 198:111, 112). You are expected to know Java, and to some extent UNIX. You are also expected to be able to write and test well-structured programs that are 5 to 10 pages long. Many students will also be familiar with the material on object-oriented software development taught in 198:113, including some UML, the use of Java Swing, and the Eclipse environment; I will try to callibrate my lectures to the average class background. Also, you need to be able to organize and write reports quickly and in clear and concise English. (This is very important: we recommend that you take a technical writing class if you need practice in this area.) You should be able to use an editor such as emacs, a document production system, such as Latex or Word, as well as a spelling correction program.

1.2 Group Project

The class will be divided into groups of 5 people. Students are strongly encouraged to form their own groups of compatible people. A suggested project will be described in class. The project will take the entire semester to complete, with major checkpoints at two to four week intervals. These include oral presentations, document submissions, and a final demonstration of the project. Sample documents from a previous version of the course will be shown in class.

Note that you will need to read and follow this document and the suggested readings on your own: the lecturer will not allways discuss the project during class.

1.3 Grades and work.

A third of your grade will be based on the group project documents and demonstrations. *You are being graded on the quality of the work you produce, not on how many hours a week you spend. Use your energy wisely!* The rest of the grade will depend on individual tests and homeworks, and your contributions to the group project. Each student must contribute to each type of activity in the project (designing, programming, code review, testing, writing, editing), in order to receive a passing grade in the course.

It is very important that project assignments be turned in promptly, both to allow you time to complete the entire project on schedule and to allow us time to evaluate your work and make suggestions. We

will be quite strict about keeping on schedule. In particular, any group that does not present a demonstration of its project on the appointed day, will not pass the course. If there are serious problems, such as a major illness, please see us as soon as possible to make arrangements about extensions. It is almost guaranteed that the computer will be heavily loaded and will crash at the most inconvenient times possible. Use autosave when editing, think before you write, code, or debug, and don't wait until the last minute to do something.

1.4 Comments from Previous Students

(These are comments from the CMU version of the course a decade ago. Ours is somewhat more reasonable but still . . .)

- Warn people ahead of time what to expect. . . a lot of work is involved
- You should only take 3 other courses when you take 431.
- Everything takes a lot longer than you think it will and even longer than that.
- This has been the most worthwhile course I have taken since I learned to write in the first grade. . . . I've learned a lot about computing, time management, people management, and my ability to stay logged in for days at a time. I even liked the course. There is only one major problem. My GPA is dropping faster than a wingless airplane because I've spent my semester working on 431.
- I have learned a lot about group projects and people in general. It was a good experience. I've heard that this phrase is a euphemism for something that is not entirely good.
- I have enjoyed this course. The experience of working in a group on a large project has been an invaluable learning experience. I also am thankful for the McDonald's down the street (that is open 24 hours) because without it, I doubt that I (or about half a dozen other 431 hackers) would have survived that last two weeks.
- **CHORUS**: This course is too much work !!!!

1.5 Working in Groups

You will find working in a group to be more difficult than programming on your own. Remember though that this is one of the main things you should learn from the course. It is very difficult to make group decisions if you cannot arrange to meet regularly. We expect every group to submit a note indicating at least one time and place where they will meet each week; those who fail to make meetings may be penalized with lower grades.

General advice on "having good meetings" includes i) starting with a written agenda, ii) appointing a meeting manager/secretary, iii) writing up and circulating the results of the meeting (decisions, to-do lists, etc.) Rotate the task of meeting secretary among team members. the

Most students prefer a democratic group organizations. In a minority of groups, those in which all students have a reasonably equal commitment to the course, this can in fact work, as the following comments indicate:

> As a group, we did not plan any managerial structure to the group, even at the request of our teaching assistant. We sincerely believed that we could accomplish our goals without strict group structure. Somehow - IT WORKED! The lack of group structure did not hinder completion of the requirements and code, and also provided a camaraderie amongst group members which might have deteriorated with strict management.
>
> Our group did manage to work well without any strict leadership structure, partly because everyone was kept pretty well informed as to the work that was due. Also, everyone participated in the writing of all the papers, no one person was dumped on. We each were given a section that was later critiqued and made a part of the total paper. I think that most groups would probably need some sort of structured management, but because of our group members' personalities, we worked well without any. . . . no one was offended if a mistake was found in either their code or

writing. . . . any problem that was found was usually helped to be corrected. This way both people benefited by knowing that the code or paper had one less problem that might have haunted us later. . . A great deal of knowledge can be obtained from other group members if you just ask all those questions that you may think are dumb.

Our group worked together very well with no formal division of management. As in any society, there are leaders and there are followers. It seems that we had a good mix between leaders and followers.

Personally I feel our group worked well together. We did not appoint a leader but there were one or two people who tended to get the position at different phases of the project. . . One person usually took charge of scribing each document and that person tended to be the leader for that document. Most of our decisions were made by all of the group. . . . One thing I would do next time around would be to set down a rule saying that everyone had to come to the group meeting with a definite progress report. The report should say I've debugged these modules and give approximate dates when the others would be finished. This would have eliminated some problems and helped us to catch others before they got out of hand.

However, most groups find that they need some sort of leader; sometimes all members try to do everything and waste a lot of time arguing over details, and sometimes members don't know what they should do unless someone gives them fairly explicit assignments. Some groups may want to have a design leader who is responsible for final technical decisions and a manager who is responsible for making sure everyone comes to meetings and does their share of the work. Since every team member must be the editor of at least one of the project documents/stages (prototype, requirements, architectural design, detailed design, test, code), this provides a natural way to rotate the leader role.

Unfortunately none of us had worked on a software team before. We were foolish and though we could use the 5-pointed star managerless technique where everyone talked to everyone and each did his equal share of each assignment. We realized pretty late in the game that this would not work. . . . any project group (task) must have a leader or manager. The sooner the group realizes this the better off they'll be. . . . As few people as possible should write a particular document. One person (definitely not the author) should edit the entire document.

For the first few documents we worked as a *group* of people, all contributing in unison. We soon realized that this was not the management method to use, and we then assigned . . . to be group manager. This was to make us more efficient in the long run ... but I found that there was almost too little communication since when we integrated the system there were a few questions of interfacing that had to be resolved that could have been settled by people communication.

Our group failed to work together. The main problem was the inability of most members to keep to a schedule. . . . I found that rotating group leaders did not work. If a group can't work democratically it should choose a permanent head.

The design process should, in my opinion, have a single person in charge of both the design and the paper. He or she should have final say in major disputes that seem to be unresolvable by any other means. . . . As many of the design decisions should be delegated as can be done reasonable. . . . he must take care to limit the discussion of relatively minor points in order to keep the meetings from getting extremely long. In a nutshell, he is referee, timekeeper, and one of the players. . . . The leader should at the close of each meeting, assign work for each of the drones including himself. . . . give the boot to nebulous group members as early in the game as possible. . . . Finally, let me say something about communication. In a word, you can't have enough.

...our group suggests that each member be made to realize how much work is expected of them and the other members must force them to live up to these preset amounts; otherwise the remaining members will be stuck with twice as much work and half as much sleep. . . . The course was truly an experience and we would recommend it to anyone!

431 has taught me that a dictator is better than no leader at all. . . . There is a lot of overhead, timewise, involved with keeping communication lines open within a group.

We did not have group meetings often enough. This caused us many problems in both the design and coding phases of the project.

I would suggest that next year's students work with people who work at the same pace. . . . "egoless everything" is a necessity. . .

...try really hard to get along well with the other group members. You will have to work really close to get a good project together and it will take everyone's work to do it.

While it is to be expected that some members of your group will be better writers, some better programmers, and so on, you are *not* to divide the labor on these grounds alone. This class is to be a learning experience, and each of you should get a substantial amount of practice in all of these areas, not just the ones you are already good at. Each person must *write* some of the documents, must *edit* one (see the next section), and must write *code*. You are *not* to turn one person into a full time typist and help-module writer just because you don't think their code will be perfect. If one group member is a fantastic writer, by all means let him edit the first document and help teach the others how to write better. He might be assistant editor for some of the other documents, but he is not to write them all. If another member is a fantastic coder, go ahead and have her be design manager, but she is not to write the code for the whole system, rather she should help the others improve their design and programming skills.

1.6 General Comments on Documentation

This course involves a lot of writing. You are not being graded directly on your writing style, but good writing usually conveys ideas more clearly than poor writing, and the process of writing down your thoughts often clarifies them. Thus, it is to your advantage to make the effort to organize your thoughts and write well, and we will make comments about your style as well as about the project you are describing.

At the beginning of the semester you should skim all of the assignments, then reread each several times before you begin to work on it. If you still have questions about the assignment after reading it carefully, be sure to ask your TA or the instructor.

Please note that document length is not equivalent to quality. If your document is longer than the suggestions given in the assignments because it is full of concrete details, fine. If it is longer because it is poorly organized and repetitious or full of five-syllable words strung together into apple-pie generalities about how modular and extendible your system will be, don't expect to get a good grade.

You are to turn in two copies of each final document. One will be read by your TA and the instructor and will be returned to you with comments and a grade; the other will be kept in our files for future reference.

Each document is to have one (not five!) editors, and each group member must edit one of the documents. Let us be very clear about what it means to edit a document. The editor has the final responsibility for turning in the document, sets deadlines for what the other members are to write, and may have to apply pressure to the other group members to do their part unless you have another manager for this. The primary function of the editor is to read the *entire* document from start to finish to make sure that everything is in place and that there are no inconsistencies between the various parts. Being an editor does *not* mean writing the whole document and does *not* mean doing all the typing and latex-ing. While the editor will probably notice and fix many mistakes in grammar, spelling, and formatting and may help out with last minute typing and formatting, this does not mean that others have no responsibility for these tasks. If the editor writes substantial sections of the document, someone else should read them to be sure they make sense. To repeat, the primary responsibility is to make sure that the document that is turned in (on time of course) has been read through by at least one person and is something that the group can be proud of. Some TAs may refuse to read documents that have not been read and proofread by the group.

To make things easier for you, we suggest you decide on a standard document format (especially for the design documents). Most of the assignments suggest an organization for the document. You do not need to follow it exactly if you have a better idea, but you should be sure to cover all the same material. **Be sure to include acknowledgements at the end that say who wrote which sections and who the editor was.**

While it is good to put some information in tables to make it easy to read, please do not waste time spending hours drawing fancy pictures.

2 Prototype User Interface

A prototype, like a model of an airplane or of a building, is something that you can build much more quickly and cheaply than the final product, and it serves two purposes: i) to convince the customer that you should be given the job since you have an attractive product, and ii) to give yourselves an intuitive feel for how the system might look and what issues you will have to face in developing it. (Sometime a prototype is used to explore some technology that looks questionable or ill-understood.)

For this reason, you will begin by throwing together the quickest, simplest version of the project you can think of. Often, when the desired functionality is clear, the central part of the task is the user interface, consisting of a command interpreter and a display manager, since this is what the users will eventualy see. But in the real world it is often hard to get the prospective user to figure out in advance what they *really* want, and a prototype demo is a good way to get feedback.

You should try implementing enough (simple) data structures and procedures so that two or three of your commands actually do something. The rest of the system can consist mostly of "stubs", that is, routines with the right number and kinds of parameters, but bodies that simply print a message like "I did what I was told", or return canned answers that only work for "rigged" inputs. If you are ambitious, you might try to prepare responses at another terminal and send them over to the running protoype in "real time". Note that efficiency is not an issue here, nor is error detection and handling, and you can expect to *throw away* most of the software you wrote for the prototype – look at it as an experiment.

You should divide responsibility for this assignment and the next between your group members. As soon as you have some idea what your system is going to do, your prototypers can begin. The rest of the group should then focus their energies on deciding in detail what services the system should provide, and how the command language and display should look.

Your will be asked to demonstrate your prototype, and others in the class may attend so that groups can learn from each other.

3 Functional Specification Assignment

3.1 General comments on the assignment

The main purpose of the functional specification is to explain *what* you are going to do for your project as opposed to *how* you will accomplish these goals. In the real world, the functional specification is the key part of the contract between you and your customer. Changes can be negotiated later and should be recorded as an appendix to this document. When writing this paper, therefore, you should keep in mind that your audiences will be a client (who in general is not an expert in computer science) and your group members, who will want to refer back to this document periodically to be reminded of what they are supposed to be doing. The fact that your document will be graded might encourage you to think and write as carefully as possible.

Discuss what you want your system to look like and produce as many examples as possible of its behavior. In your group meetings you can take turns playing different roles. One person could be the system user, one could play the system, one could be system implementer and make sure the "system" is feasible, one could be the group facilitator. In the modern parlance of UML (which will be covered in lecture), the majority of this document will consist of presentations of *"use-cases"*. The paper should be 15-25 pages; if it is much longer or much shorter you should consult the instructors.

3.2 Typical contents of a functional specification

Title Page

This assignment, *as well as all others throughout the semester*, should be turned in with a title or cover page that includes the following information:

- A title that identifies the document (such as "Functional Specification of the ALPHA Calculator Development System")
- The name of your group (such as "Great Hacks, Inc.")

- The people who contributed to this document, identifying the editor.
- Course name, instructor's name, TA's name, date

Functional Summary

In about 2 pages, you should briefly summarize the project you are working on. Give your system a name. Describe what it does and who will use it. What needs will your system satisfy? How will it help the users? Outline the most important features of your system. Describe the physical environment in which your system will be used, including any other systems with which the new system will interface. Are there any important performance goals for your system: time or space efficiency, security, or reliability? Any system (in)dependencies?

Details Concerning System/User Interactions

The main point of this section, which should be approximately ten pages, is to describe the functions that the system will perform *from the point of view of the system user*. You need to cover the kinds of inputs your system expects, the actions it will take on both expected and unexpected inputs, and the types of outputs that the user will see in these cases. Part of this section may eventually be developed into a user manual. First we'll consider the content, then the form, of these descriptions.

The inputs, state changes, and outputs should be described in reasonable detail. You need not stick to the exact wording of messages, but you should make definite statements. You can negotiate major changes later. Describe what legal values or ranges your system will accept for inputs, what precision or accuracy constraints you will follow, and how you will handle errors.

- Will your inputs be mostly commands typed on a terminal? What kinds of terminals? You should find out what kind of terminal will be used for the final demonstration and plan a system that will run on that type of terminal and at least one other type to make sure your design is flexible enough for extension. Will your program need to read the system clock or look at files or read signals from external meters?
- What functions or transformations does the system perform from the user's point of view?
- Will the outputs be short messages to the terminal? Actions such as writing reports or system backups on the line printer or on files or tapes? Control actions such as shutting off valves? What kind of error messages will you give?
- What kind of help facilities will you provide for system users?

Your functional description should employ some of the following techniques:

- **Scenarios and Use cases**. A very important part of the description are sample interactions with the system in the form of a transcript of a dialogue between a user and the system. Use upper and lower case or some similar convention to distinguish between what the program types and what the user types. (Examples of this will be discussed in class.) You will be giving a demonstration of your system towards the end of the semester, and your goal in that demonstration should be an elaboration of the dialogue described here.
- Domain model, identifying entities, relationships, processes, constraints, etc. (expressed in UML – also described in lectures).
- **Glossaries** Be sure that all specialized terms are defined in the body of the document or in a collection at the end. This could include computer science terms that you wouldn't expect you client to know or application terms (from accounting, geology, or whatever) that programmers working on the project might not understand. (Disambiguating such terms is an important part of practical requirements engineering.)
- **FSM model**. If your system can be thought of as being in a number of states, you can draw a finite state machine model (FSM). The different states might correspond to different prompts for user response that are shown in your dialogue.

- **BNF specification**. If much of your system involves commands specified by the user, write a BNF describing the syntax of these commands. Similarly, if some input or output of the program is structured,

Feasibility

Make sure your project has a chance of being completed in a semester. What are your preliminary thoughts on how you will break down the different features of the project? What are the major classes of functions and the relationships between them? For example, are there fairly separable components such as an editor, move checker, function simulators, database access functions? Think a little about your main data structures and algorithms, and spend a page or so discussing possible implementations. Don't waste a lot of time fighting over details, but be sure to ask your TA or the instructor if you aren't sure how hard something will be.

Our System, Plain and Fancy

Predicting how much can really be accomplished in a finite period of time is often tricky. It is important to decide what a bare bones version of your dream would involve. Describe, in one or two pages, the composition of your skeletal system. What functions will not be included, and why? Cover yourself by making sure that reading between the lines doesn't make your customer think that more is being promised than you plan to deliver. This basic system should provide the most important functionality, from the point of view of the user, so you may need to interview him/her. And a basic system might provide a less fancy interface to the functionality being offered. Now sketch out which features will be added as time permits. Try to organize the bells and whistles into packages that could be added independently or in some predictable order. Also think about the order in which parts of the system can be dropped so that you can retreat gracefully if necessary.

A good way to describe your series of systems is in the form of user reference cards. You should make a series of three or four cards outlining what is contained in (or added by):

- a kernel system with absolutely minimal features (that you are positive you can finish in 3/4 of a semester)
- a cheap but usable system (that you expect to finish)
- a standard system (that you have a good chance of finishing)
- a super system (that you probably won't be able to finish but would like to; don't make it impossible, though)

Important! Since most software projects in the real world are late, or never completed, it is very important to be able to deliver to the customer **something** by the deadline. It makes sense therefore to make the kernel that part of the system which would seem the most useful/important from the point of view of the customer – i.e., the part which gives the biggest "pay-off" for the effort invested. There is therefore a delicate balance between making a system "user friendly" (it may not be used at all if it is very unfriendly!) and adding more functionality. You should try to get feedback from your propsective users (and, if necessary, use your judgement) in deciding these matters.

Summary

Don't drop the reader off a cliff at the end of your paper. It's been a long time since the beginning of the paper. Restate the main points you want remembered.

Acknowledgements Write down the authors of individual sections, the editor of the whole paper, outside consultants, etc. (Do this for all future papers as well.) This will help people figure out who to ask for more details.

Optional Sections

A real functional specification would include a number of items that you may wish to skip at this point. Some of them require more experience to predict and may be covered later, some are not relevant to all projects, and some have already been discussed. Use your good judgment.

- Make some general statements about the performance goals for your system. What are your goals for system run time and primary and secondary memory use? What kind of reliability will you guarantee? Security of information? What kind of performance trade-offs have you decided to make?

- Can you say anything about compatibility with existing software or hardware? What about an installation agreement? Maintenance contract?
- What resources are to be committed to the project? Who are the people on the project? Their skills and background? What is the promised delivery date? How much computer time and space will be used to develop the project?
- What publications will be produced? Who are the intended audiences?

3.3 Some comments by previous students

Writing down the functional specifications is fundamental to having a good start on a well-designed project. No matter how much you discuss something there will be loose ends and not until you write your ideas down will these unresolved parts become visible. The most important sections of the paper are those dealing with the functions of the system. You need to understand thoroughly "the transformations from the user point of view." You should not have any vague points unless you absolutely cannot help it. If you do, it will be very hard to design your project because you do not know what you are trying to achieve. Even though the emphasis of this paper should be what your total system will provide, you should also begin to think a little about the implementation.

In some respects the Functional Specification was the most useful of the documents. It forced the group to adopt a rigid management structure (we quickly found that five people writing the document causes tremendous amounts of bickering) and outlined the goals of the project. It also provided a first look at the more serious problems which the project would have to overcome.

Make sure that you write down the things that must become gospel like the format of input or the syntax of a language. Make sure that you give enough detail in these areas so that the document gives a unique answer to a question, two people may give conflicting information.

Quite frankly, we did rather poorly on the Functional Specifications paper. That error hindered us for the rest of the term. Our main flaw was not an inability to agree on certain issues; it really centered on not writing down all decisions, (large and small). As a result, I would strongly urge future groups to do so.

When we completed the project . . . we were amazed by how similar what we had implemented was to what we presented in the functional specification . . . . First, it is important to do a lot of thinking before you actually say what the system is going to do. Consider *all* features that you might want to add at a later date, include them in the functional specification so that when you design the system it will be designed with those features in mind. It might be interesting to look at another group's functional specification and see if you have left out any major areas.

One major problem we encountered was in breaking the project into equal parts. In writing the Functional Specifications, we didn't think about the work involved for each part. Consequently, when we broke up the project into 5 parts, they were not equal parts. Some people did a lot of coding while others had almost none.

4 Design Document

4.1 The Design Process

The design of large systems is still an art. The end goal in the design process is a *detailed design document* that is complete enough to be a reference from which any competent computer scientist can produce code, test plans, or a user manual. (This document should eventually evolve into the code and system maintainers' guide.)

4.1.1 Architectural Design Specification

An intermediate goal is the *architectural/overall design specification*. This can be considered a draft of the detailed design document with some of the details missing. At this stage, the design should be complete in the sense that all types of inputs and outputs and transformations and special cases have been considered and the system decomposition has been decided. Interfaces between modules should be clearly defined. This overall design will be evaluated, and you will receive feedback on it, so that possible improvements can be made.

Once the overall design has been written down, everyone in the group should read the entire document to make sure they understand the design. Individuals should be assigned responsibility for particular modules and should design these in more detail. Make sure the division is equal in the amount of work involved. Try to estimate the number of lines of code and the difficulty of each of the modules. You may give people with less software experience the easier modules, but make sure the division is not extremely unequal. Each person should carefully review at least one other person's section (and all sections should be reviewed by someone other than the author). Pick the one(s) with which your module has the strongest interaction(s). Based on the examination of the module interactions, you should prepare a set of *interface definitions*. Each module M should have a set of *public exports* to the outside world that are all any other module ever needs to know about module M. The interface definitions should be complete, including type declarations and procedure definitions (function names, parameter types and orders). [In the object oriented world, a module specification corresponds to the public parts of a class definition, an interface or a package. We will talk more about this in lectures.]

4.1.2 Detailed Design Document

The final step is then to prepare the detailed design document. Most of the work will be done by individuals filling in the details of the specification of their modules, but you must review the document as a whole to make sure people are using the same interface definitions and to make sure that everything has been covered and that the parts of the document hang together.

4.2 Getting Started

Review the functional specification from the point of view of the implementer. Make sure that the user interface is well-specified by a BNF or equivalent. What types of services does your program need to provide? Try drawing a data flow diagram of your system, and try constructing a structure chart. Can you split up the functions and data objects into abstractions or modules? Once you make a tentative division of responsibility, play the human computer game. Assign each person to a module or set of modules. Follow the sample dialogue in your functional specification and extend it to include a dialogue *between* the modules in your system to see *how* you will accomplish the actions you have promised. After you are satisfied with your basic modular decomposition, have someone play user (perhaps someone from outside your group) and give some unexpected inputs. Think about what kind of responses you want your system to give and which module(s) will be responsible for generating them. Is your proposed help facility sufficient? Finally, think a little about how your current set of modules could be extended to handle your deluxe system. Also think about what debugging aids you might want to build into the system for the benefit of the implementers.

4.3 Writing the Document

Remember! Every document should state, in the acknowledgements and/or the title page, the author of each section and the editor of the document.

First of all, remember Brook's comment *The Mythical Man-Month*, page 165

> Most documentation fails in giving too little overview. The trees are described, the bark and leaves are commented, but there is no map of the forest. To write a useful prose description, stand way back and come in slowly.

Your goal should be to provide a top down description of your program in a form something like the following. You probably won't be able to *design* your project completely top down, but such a description should be a good explanatory and reference tool.

Give an overview of your implementation, including its structure and philosophy. Motivate design decisions where necessary and show how they are reflected in the individual modules and in the interactions.

For each level, you should cover the following:

- **Abstract** – What services does this program, module, routine, etc. provide to outsiders?
- **Implementation Documentation** – Are any special instructions needed for the system user or for the writer of other modules?
- **Design** – What is the basic design of this module or routine? How are the design decisions reflected in the submodules or routines which make up the whole? How are the pieces combined to accomplish the main function described in the abstract?
- **Exports** – What routines does this submodule make available to other modules? What type/class declarations? What constants?
- **Imports** – What procedures defined in other modules are used? Beware: a procedure, function, or type cannot be both an import and an export of the same module! During the early design stages you may do this by hand; later you will want to use editor or text-formatter commands to do this. Thus, this information might be part of an index rather than distributed throughout the document.
- **Input/output**– Make sure all necessary actions are specified.
- **Subparts** – What are the names of the submodules, routines, and/or data abstractions that make up the module or submodule?
- **Pre and Post conditions** – What are the pre and post conditions on the main routines in your module? What are the important invariants on your data structures? Answering these questions will help answer the next two questions and should make your interfaces work properly.
- **Error Handling** – What is the range of legal values? What happens when other values are found?
- **Test Cases** – List all your fiendish ideas for cases that might break the system or module or routine. If you do a good job here, you will be more sure of having a good design and most of your test plan will be already written.
- **Concrete Implementation** – For (sub)modules that are data abstractions, give the concrete representation (data structure as diagram or class/type declaration declaration) here; the access functions will be listed as subparts. For routines, this is where the pseudo-code goes. Define any technical terms relevant to the module.
- **Side Effects** – Hopefully there aren't any, but if there are, you'd better make them explicit.
- **Miscellaneous** – Just in case you think of anything else (for example, an estimate of how frequently various routines might be used or how much code will have to be written), throw it in!

In addition, there are a few more global questions you should answer, separately or within module descriptions.

- What are the most likely modifications that a client might ask for or improvements that you might want to provide? How easy would it be for you to make those changes? What modules would be affected, and how extensively? Would conflicting design considerations cause some modifications to be infeasible or to require extensive changes?
- What are the implications of alternate design decisions on higher/lower/same level modules?

- Do you want to add anything to your design to make debugging and testing the system easier? This may be discussed separately or included in the discussion of some modules.
- Management Issues: What implementation strategies will you use? Who will be responsible for the further design and coding of the pieces? Estimate the time needed for the *development* of the major sections of your system based on the estimated size and complexity. Remember to concentrate on the critical pieces of the system.
- Have all interfaces to secondary storage or hardware devices been discussed?
- If appropriate, use diagrams of control flow or data flow for the systems or individual modules within the body of your description. Don't forget to label diagrams.
- After you have completed the interface definitions, add some cross-referencing information. You will often need to be able to find all places from which a routine is called. Don't do this by hand; try to use word-processor indexing commands or at least some editor commands. Unix also has cross-referencing tools for programs.

Obviously, not all categories are needed at all levels. For access routines, for example, all that is needed may be a good function declaration. (Also, please don't call all of your procedures modules!) Where appropriate, write your description in a formal high-level language. There is no need to repeat things in English that are obvious from the type declarations. In the overall design, you may make statements like "X is a set" or "search for y in X" rather that saying that X is an array and going into the details of the binary search. Try to use meaningful function names, after the group has agreed on naming conventions.

Checking the Interfaces

To check the interfaces in your project, use the following procedure. For each of the five or so major modules, you should make (or already have) a list, probably alphabetized, of all types, functions, and procedures that are exported. For each type, the declaration should be written out. For each function or procedure, there should be a header (names and types of parameters and values returned) that will pass through the compiler. For each type there should be comments about whether the implementation is hidden and what or where the access functions for the type are. Pre and post conditions for the procedures should also be included in comments. The designer of every module that imports anything from this list should sign, in blood, a statement that they won't need to import anything not on the list and that they are happy with the current headers and declarations. You should also record the names of the modules that use each types or routine so that if changes are considered, all people concerned can be consulted. It is recommended that you follow the same procedure for at least one more level within each module.

Detailed Design Document

The detailed design document should include answers to all of the questions for as many levels as possible. You do not have to write the code to do this, but there may be some modules for which the clearest way to explain the function is to write an outline of the code. This is acceptable if the code is written in terms of calls to other high-level functions no bit-twiddling and is easy to read. Update the sections describing coding responsibilities. A table of contents and an index are necessary in this document. If your document seems to be getting too long (more than 60 pages) consult your TA.

Good luck!

4.4 Comments from previous students

> The (architectural) specifications were extremely helpful when designing the code. Our specifications were detailed enough that attempting to write a piece of code for a module could almost be taken directly off of the specifications. Writing the design specifications twice helped us a great deal. Many things were pointed out to us that needed improvement, and the second set of specifications gave us time to think things out a little longer.

> The overall and detailed designs were the most important and effective aspects of the project. Without the detailed design as a reference, coding would have taken significantly more time and integration of the modules would not have proceeded as smoothly as it did. ... We did define

most of the low level routines, and it was to our advantage, as almost all other routines depended on the existence of low level functions.

We should have used a more consistent and reliable method of updating our database. One suggestion which seems like a good solution to the problem is to use access routines every time the database is used. ...I think that future 431 students should be required to use access function or to present overriding reasons to the contrary. Put it down as a mark for structured programming/the american way/etc.

Every time I violated the principals of abstract design, I suffered. Abstract data types, modular decomposition, and information hiding can really work.

The design process was definitely a major factor in simplifying the debugging process. It is interesting to note that we had almost no interface errors. We attribute this to good communication within the group.

If we had it to do over again ... we would use the Pascal utilities Include command. Many of our problems in debugging had to do with mismatched types. These came about as a result of sloppiness in updating changes, poor file control, and poor communication. ... Our design was an incredible aid in coding and debugging. It was exactly where we designed sketchily or improperly that we had the most problems. Our interfaces were not always clearly defined, which led to type errors.

...the overall design should be geared more toward modularity and expandability than toward speed. This was a major obstacle in our agreeing on a design, but I think deciding on a modular design paid off. The design bugs we found during the implementation might have prevented us from completing the project according to the specifications if we had not been extremely concerned with expandability. It seems that efficiency should be worried about later, at least for one's first design.

5 Testing Plans Assignment

5.1 Overview

Testing is an important part of a software project. The software team must therefore carefully plan and document the order in which modules are to be integrated and the order in which the individual modules are to be completed and tested in isolation. Testing and debugging methods must also be agreed upon, documented, and eventually carried out. The grade you get on the test plan is tentative; a final grade will be assigned after you complete the testing process.

The purpose of a test plan document is to convince the management (in this case, the teaching staff) that a feasible test plan has been designed and also to provide the project members with directions for the testing phase of the project. This assignment outlines some constraints on the test plans. Several stages of tests must be scheduled, and several testing procedures must be explored. Scheduling is required for unit tests, integration testing, functional testing, performance evaluation, and an acceptance test (the demonstration). These tests must include practice in the techniques of walk-throughs, extensive logic testing, input/output testing, and optionally verification. The following sections give details of the contents and style of the test plan document, and the final section lists some reading material that should help you design a better test plan.

5.2 Designing the Test Plan

The test and evaluation plan should contain the following components: statement of objectives and success criteria, integration plan, testing and evaluation methodologies, and responsibilities and schedules. These components are described in more detail below.

Objectives and Success Criteria: The test plan document should contain a statement of the overall testing objectives and the objectives of the individual tests that are planned. Objectives might include testing the program logic of individual modules, validating the functional performance, or testing the hardware

interface. Success criteria might be finding a certain pattern in the number of bugs detected, exercising some percentage of the code, or analyzing performance to a specified accuracy.

Integration Plan: An important decision to be made about testing is the order in which modules are to be combined and therefore the order in which they will be tested individually.

- You must plan for individual module tests, the combination of modules during integration testing, a functional test, and an acceptance test (the demo). You should also plan for an evaluation of the runtime performance of your system.

- One of your major goals must be to get *something* working by your deadline, even if it isn't the full system. Go back to your specification and refresh your memory on what the kernel system was to look like, as well as the increasingly more complete systems. Determine what needs to go into a skeletal system and then schedule the testing and integration of those components first. (You may in fact discover that there is a better sequence of plain-to-fancy stages, now that you know about implementation problems. Make this a part of the document.)

- You shouldn't expect to be able to debug separately your 5 portions and then integrate them one by one in 5 steps (let alone throw them together all at once). Instead, you should identify IN EACH MODULE the most indispensable portions (types, procedures/functions), and test and integrate these first, thus bringing up a kernel system. You can then iteratively add new parts to the system and continue testing the results. This incremental approach ensures that you will not be snowed under an avalanche of bugs/bugs caused by fixes to bugs/... and that you will be able to have at least something running.

  Concerning the above, note that even within one method, (as it was described in your design document), there may be an essential component without which the rest of the system cannot possibly work. You may have to test and integrate this portion first, and add the remaining code to the procedure later. This can refer for example to error-checking code, or other bells and whistles, whose skeleton may be hidden at the beginning using comment symbols.

- The test plan document should describe and defend your integration method. The "big bang" method of integration (putting all the pieces together at once and crossing your fingers) *is not* acceptable, but many variants or combinations of top down and bottom up integration are acceptable if convincingly defended. Define any uncommon terms used in your document; your TAs (and your managers in the real world) may not know what "modified sandwich testing" means unless you explain it.

- It would be particularly helpful if you summarized your plan using diagrams and/or tables.

Responsibilities and Schedules: The test plan should provide a schedule describing dates and responsibilities.

- First, determine an order in which to perform integration and functional tests. In scheduling, you should make use of the dependency graphs based on the external functions that the modules require. You should explain convincingly why your ordering makes sense, and why it will produce *something* that works in time for your demo, even if all the fancy features have not yet been debugged.

- Next, this order should be used to determine the order in which the individual modules are to be tested. Determine the dates by which tests are to be completed and the individuals responsible for the testing. Prepare a master test plan schedule and include it in the document.

- Finally, you must define a monitoring procedure to ensure that tests are designed and carried out on schedule. Someone will have to keep a notebook and report lack of compliance to the other team members. Similarly, there must be a procedure for reporting and correcting bugs. In this document, you are to describe the monitoring, reporting, and correcting procedures that you will use. In your final evaluation document, you will discuss how well they worked.

5.3 Writing the Document

The test plan document is to contain an introductory section that summarizes the whole document in a page or two and discussion sections that cover the plans in more detail. A total number of pages between 15 and 25 is about right.

Remember that this document is intended for several audiences. The document should be organized so that it is easy to find schedule summaries and monitoring plans and easy for individual team members to find directions on their personal responsibilities. A table of contents will facilitate finding this sort of information and should therefore be included. The test plan should not be very long; if you find that the writing of the document is taking significantly longer than the design of the test plan, consult a member of the teaching staff.

The introductory section of the test plan document, which should only be about three pages long, should include the following:

- Overall objectives and success criteria
- Summary of the integration plan – a list of tests and dates and people responsible
- Summary of the module-to-test-technique mapping for the four required testing techniques
- Summary of the monitoring, reporting, and correcting procedures

The discussion sections should contain:

- Defense of the integration plan (about half a page)
- Details of the tests with objectives and success criteria for each test or group of tests (assuming you have proposed most of the tests in your design document, there should not be more than 20 entries in this list, and each entry should be less than one page long; if you did not have good tests in your design document however, you should include them here)
- Details of monitoring, reporting, and testing procedures (a page or so should suffice)
- Details of individual team member assignments (just a page or so, this is basically a cross-reference list)

5.4 Comments by previous students

> Some members of our group didn't think it was worth specifying what levels of what modes we were going to try to bring up first, second... If we had done this then we could have concentrated on the parts significant to the first level. It ended up that when the integration began, whole procedures were unwritten that were needed by other modules, names and parameters didn't match up, etc. ... I can assume there was a lot of time wasted writing code for modes that were not fully implemented in other modules!
>
> We proposed to implement only a few functions in various categories, but instead actually tried to implement all. As a result, it was difficult to test the integrated system. It would have been better to try running a small system which handled all modes, rather than a large one which was buggy in even Immediate mode. ... Our design was generally adaptable to the smaller, more complete system and certainly wasn't responsible for hindering us from building it.
>
> Note that the test plan had a very positive effect on the testing process. This was not so much because we sat down with the plan and used it to run our tests, but because it forced each group member to write down all error conditions that their module would test.
>
> Since none of the other modules were ready and the tester of my module was busy with his own code, I decided to do walk throughs of my code. This proved to be extremely valuable especially when I waited about 3 days before doing another walk through. I caught many small errors and inconsistencies. This significantly reduced the amount of machine time spent debugging. I also think having other people's specifications helped. Once movement (her module) was executed, it ran. I never had to change one piece of code concerning the legality. The bugs resulted from

. . . failure on my part and the database programmer to agree on what a routine was supposed to provide.

Debugging was quite a bit harder than anticipated. . . . Misuse of routines seems to be by far the most common error (causing string storage not to be reclaimed or overwriting the wrong strings).

One clear problem (during coding and testing) was communication. We should have continued to meet even after the designing process was over. Many problems encountered were due to one person's assumption of another module, or changes made in one place and not in another.

The idea of interactive testing should be pushed as it is a very efficient manner with which to test plus if those people testing come up with test cases that they had not previously thought of, they can run them spontaneously without having to re-compile a fixed program.

Modified "sandwich" testing worked well for our project. It allowed some modules to be tested independently of other, incomplete, ones. Later, when all the pieces met in the middle, there were very few module bugs. This made it possible to find and correct interface and overall system bugs. . . . Testing should include outside tests by another group.

(Our) order of integration proved to be very effective. We debugged using stubs and drivers as long as it seemed efficient to do so, and when we needed them, the other modules were ready to aid us in further testing.

We were surprised when the program grew faster as we were fixing bugs than it had when we were writing the original version. We were not prepared for the effect of the "patch it quickly" attitude on our program. As we put in hack after hack to get the bugs out, some of which were put in to undo side effects of other bug fixes, the program ballooned. . . there were more bugs than we had expected. Even in the simplest code, the parts that we thought couldn't have bugs because of their simplicity, we found bugs in nearly every procedure.

If one does not understand correctly what the program is supposed to do, one can write test cases that produce correct outputs according to the program, but not according to what the program is supposed to do . . . a program can behave as the writer of the program would expect but quite differently from what the person who wrote the specification would like.

6 User Manual

This document should be a self-contained description of how to use your system. A user manual should be a polished, professional piece of technical prose that a software company is proud to have accompany one of their products. (And it is a handy accomplishment to show off at job interviews when you finish university.)

The document should have a structure that is evident both to someone reading it straight through and someone looking for a particular topic or fact. A table of contents is required, and the organization it reflects should be considered carefully. An index and appendices might also be helpful.

Think about who your audience is, and what they know or care about. You may want to talk to two different audiences: installers and users. This is fine, but be sure to address the different types of people in different sections of the document. Since the user is the intended audience of the document, it is less awkward to write "first you..." rather than "first the user...." Better yet, say "First do x." Test out the user manual on your friends.

Remember, the document should be completely self-explanatory. Do not assume the reader has seen your functional specifications. You may of course edit in sections of prose from your previous documents. Do *not* discuss any implementation unless it directly affects the user's interface to the system.

Your user manual should be no longer than 20 pages. A short command summary or pocket reference card will be required.

Your document ought to cover the following list of topics. The exact order in which you present material and whether certain topics are combined should be dictated by your particular project and your own style of writing.

- Introduction – a concise statement of what your program does, possibly including motivation and philosophy. State what the reader should know before reading the rest of the document and where to find information about any assumed prerequisites.

- How to use your system – an overall description of the style of user interaction, device constraints, and any arbitrary design choices you made that the user ought to know about.
- Detailed system operations – an organized list of all the user commands and when they are appropriate; some examples might be helpful. A novice section and an expert section are also possibilities.
- Error recognition and handling – what to expect and what to do.
- An extended example is a must – show exactly what the user typed in (and why) and what the computer typed out and why. Use some convention such as upper and lower case to distinguish between user and system statements. This is a very effective teaching technique, and will help you prepare for your demo as well.
- A list of known "features" and deficiencies.

Everyone complains about how cryptic current computer manuals are; this is your chance to show us what a *good* manual looks like!

7 Demonstration

The scheduling of demonstrations will be different for each class. The following describes how it was done at CMU:

The last two weeks of class time are reserved for project demonstrations rather than lectures; we will meet in the small room terminal room. We will connect all the terminals in a talk ring so that everyone can see what is on the screen. Each group will have about twenty minutes to demonstrate their system. You should prepare about 15 minutes of material that exhibit the best features of your system and allow about 5 minutes for questions and feedback from the audience. If your system has been properly debugged, you can let the audience tell you what to type to show off your error handling and user help facilities. Everyone in the group should participate. You can have each person explain their part of the system or have one person typing while another is talking. You are required to give out a short description of your project or a user card. Because the time is so short, it is important to practice what you are going to say and do. You should give at least one practice demo for your TA a week or so beforehand.

7.1 About Reference Cards

There is not enough time in the course to require a user manual from each group. Instead, you must prepare a brief summary of the user interface to your system, formatted in a way that a user with some experience can easily find the information he needs to use the system. Such a card is normally formatted to fit in a shirt pocket. You can do this by such tricks as using the Scribe multi-column facility, or by photo-reducing conventionally prepared text, or by copying on both sides of the paper.

For ideas on what the card might contain, look at the emacs reference card, any assembly language reference card, and the suggestions about user manuals given earlier in this handout. A complete list of available commands and their meanings, and a sample dialogue, are "musts" for the reference card.

7.2 Student comments

> PRACTICE YOUR DEMO BEFOREHAND. Write up a script of what works on your system and what doesn't so that you will not be thrown into PasDDT by accident. It really looks bad after a whole semester of work.
>
> Keep at least one old EXE file around before your Demo in case of emergency. It is also a good idea to keep around two copies of the sources in case something mysterious happens the morning of your Demo.